Chapter 3
Software Product Quality Models

Rudolf Ferenc, Péter Hegedűs and Tibor Gyimóthy

Summary. Both for software developers and managers it is crucial to have information about different aspects of the quality of their systems. This chapter gives a brief overview about the history of software product quality measurement, focusing on software maintainability, and the existing approaches and high-level models for characterizing software product quality. The most widely accepted and used practical maintainability models and the state-of-the-art works in the subject are introduced. These models play a very important role in software evolution by allowing to estimate future development costs, assess risks, or support management decisions. Based on objective aspects, the implementations of the most popular software maintainability models are compared and evaluated. The evaluation includes the Quality Index, SQALE, SQUALE, SIG, QUAMOCO, and Columbus Quality Model. The chapter presents the result of comparing the features and stability of the tools and the different models on a large number of open-source Java projects.

T. Mens et al. (eds.), *Evolving Software Systems*,
DOI 10.1007/978-3-642-45398-4_3,

3.1 Introduction

The need for measuring the quality of software products is almost as old as software engineering itself. Software product quality monitoring has become one of the central issues of software development and evolution. Both for software developers and managers it is crucial to have clues about different aspects of the quality of their systems. The information is mainly used in making decisions during software evolution (e. g., to start a refactoring phase or reimplement a system because of wear out), backing up intuition, estimating future costs and assessing risks.

A large number of quality models, measures, and approaches have been introduced in the past. These software quality assessment models belong to one of the following types:

1. *Software Process Quality Models* – the idea behind these models is that they measure and improve the software development process. These models are based on the assumption that better development processes lead to better quality software products. These models make their estimation based on different process metrics (e. g., defect removal efficiency, percentage of management effort for a given project size, average age of unresolved issues). Some of the well-known process quality models are SPICE [259], ISO/IEC 9001 (Quality management systems – Requirements) [427], and Capability Maturity Model Integration (CMMI) [184].

2. *Software Product Quality Models* – these models measure the software product itself. They measure different kinds of source code metrics (e. g., Lines of Code, McCabe's cyclomatic complexity, coupling) and combine them somehow to assess the quality of the product. Early quality models are McCall's [574] and Boehm's [122] models followed by the standard ISO/IEC 9126 [422] and its successor ISO/IEC 25000 (SQuaRE) [425]. Many practical product quality models have been derived from these standards since then (e. g., ColumbusQM [63], SIG [387], SQALE [516], SQUALE [619], QUAMOCO [905]).
3. *Hybrid Software Quality Models* – these models combine the previous approaches: they calculate both product- and process-based metrics to assess the quality of software, like in the work of Nagappan et al. [630]. Particularly, they added line changes, code churn and other process metrics to software product metrics and built a hybrid model for post-release failure prediction.

This book chapter deals only with the second type of models that assess the software quality based on software product metrics.

Even though early product quality models have appeared in 1977, right after the introduction of the first source code metrics, the explosion of new practical quality models has started after 1991 with the appearance of the ISO/IEC 9126 software product quality standard (see Figure 3.1). This standard defines six high-level product quality characteristics: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. The characteristics are affected by low-level quality properties, that can either be *internal* (measured by looking inside the product, e. g., by

analyzing the source code) or *external* (measured by execution of the product, e. g., by performing testing).

In the context of software evolution, which is the focus of this book, maintainability is probably the most attractive, observed and evaluated quality characteristic of all (discussed in more details later on in Section 3.2). The importance of maintainability lies in its very obvious and direct connection with the costs of altering the behavior of the software [62]. Although, the quality of source code unquestionably affects maintainability, the standard does not provide a common set of source code measures as internal quality properties. The standard also does not specify the way how the aggregation of quality attributes should be performed. Thus it offers a kind of freedom to adapt the model to specific needs.

Many researchers took the advantage of this freedom and a number of practical quality models have been proposed so far [1, 51, 63, 68, 387, 516, 619, 905]. Most of the models discussed in this chapter share some basic common principles:

- They extract information from the source code, therefore they assess quality properties related to software maintainability. However, we often refer to these models as quality models as they define quality to be the maintainability of the code.
- Each of them uses a hierarchical model (e. g., Figure 3.2) for estimating quality with some kinds of metrics at the lowest level. In the case of each considered source code metric, its distribution over the source code elements is taken. Either the whole distribution, or a number (e. g., average), or a category (based on threshold values) is used for representation.
- The number or category is aggregated "upwards" in the model by using some kind of aggregation mechanism (weighting or linear combination, etc).

Many of these practical quality models have been implemented and integrated into modern tools supporting software evolution. They allow a continuous insight into the quality of the software product under development. Moreover, many other direct applications of these models exists. Besides system level qualification some of them provide a list of critical elements that programmers should fix in order to improve the overall maintainability of the source code. As an example, Section 3.2.4.5 presents a drill-down approach demonstrating a sophisticated technique for deriving maintainability values at source code element level. Another popular field of application of these models is in the cost estimation of future development efforts [62].

This chapter is organized as follows. Section 3.2 gives a historical overview of software product quality measurement starting from the first software metrics through simple metrics-based prediction models and early theoretical quality models to the state-of-the-art practical quality models. In Section 3.3, an application of practical quality models during software evolution is introduced. Section 3.4 collects and describes some of the available tools implementing the modern practical quality models. Then, in Section 3.5 we evaluate the introduced practical models and their implementing tools. First, we compare the models behind the tools based on a set of evaluation criteria, and afterwards we present our experiences of using

the tools by analyzing different open-source Java projects. Finally, we conclude the chapter in Section 3.6 and list some of the future research directions in the field.

3.2 Evolution of Software Product Quality Models

The need for measuring the quality of the software products has almost the same age as software engineering itself. The measuring approaches have gone through a rigorous evolution during the past 50 years. The history of software product quality measurement is presented on the timeline in Figure 3.1.

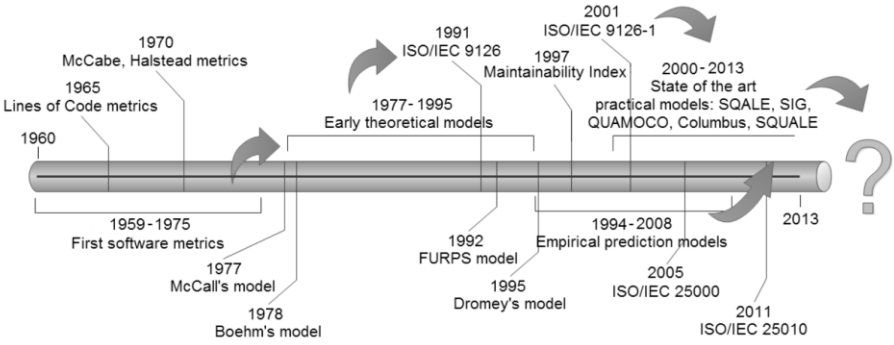

Fig. 3.1: The history of software quality measurement

The first tools for assessing product quality were simple metrics like Lines Of Code, McCabe complexity or Halstead's metrics. They started to appear from the mid 1960's. The growing number of metrics has inspired the appearance of the early theoretical quality models like McCall's [574] or Boehm's model [122] at the end of the 1970's. They all tried to capture high-level quality properties based on a hierarchical model. In 1990's all these theoretical models have been merged into the robust ISO/IEC 9126 [422] software product quality standard that had a huge impact on further quality models. The standard has been revised resulting in a new edition in 2005, marked as ISO/IEC 25000 (Systems and software Quality Requirements and Evaluation – SQuaRE) [425].

Another branch of quality assessment approach that started from the mid 1990's is the development of empirical prediction models using software metrics as predictors. These approaches try to predict software quality by using different techniques like regression [663], neural networks [952], or Naive-Bayes classifiers [869] based on empirical studies. One well-known such model is the Maintainability Index.

To overcome the complexity and lack of application details of the ISO standards as well as the hard interpretation and explicability of the empirical prediction models, a whole set of new practical quality models have been introduced in the past

few years (e. g., ColumbusQM [63], SIG [387], SQALE [516], SQUALE [619], QUAMOCO [905]). Most of these models follow the structure of the ISO standards but also define concrete source code metrics and algorithms for aggregating them to higher levels of the hierarchical model. The problem of the hard interpretation of the results has been addressed by utilizing so-called reference systems (benchmarks) that serve as the basis of the qualification. As another possible solution the concept of *technical debt* [146] has been introduced. This term was coined by Ward Cunningham to describe the obligation that a software organization incurs when it chooses a design or construction approach that is expedient in the short term but that increases complexity and is more costly in the long term.

Although the models share a lot of properties, they also differ in many aspects. It is a very interesting open question if these practical models can be unified and merged into a common standard like it was done with the early theoretical models. Our vision is that these practical models can be merged into a common standard in the future to form a whole new direction of software quality assessment.

This section gives an overview of the evolution of software quality measurements and approaches starting from the first software metrics through simple metrics-based prediction models and early theoretical quality models to the state-of-the-art practical quality models. At the end of the section we also present some of the current applications of the existing practical quality models.

3.2.1 Software Metrics

Although the first dedicated book on software metrics was not published until 1976 [326], according to the software metrics roadmap of Fenton and Neil [290], the history of active software metrics dates back to the mid 1960's when the *Lines of Code* (LOC) metric was used as the basis for measuring programming productivity and effort. In the late 1960's LOC was also used as the basis for measuring program quality (normally measured indirectly as defects per KLOC). One of the first prediction models was presented in 1971 by Akiyama [10] when he proposed a regression-based model for module defect density prediction in terms of the module size measured in KLOC.

Starting from the mid 1970's an explosion of interest arose in the measures of software complexity (pioneered by Halstead [361] and McCabe [573]) and measures of functional size (such as function points pioneered by Albrecht [5]), which were intended to be independent of the programming language of choice. The Halstead complexity and McCabe cyclomatic complexity became the main predictors of different quality aspects and effort estimation. Early theoretical quality models (McCall's, Boehm's, etc.) have started to appear also in the mid 1970's and were based on software metrics.

With the appearance of new programming paradigms such as object-orientation a whole new set of metrics have been developed. The most well-known metric suite for OO systems was introduced by Chidamber and Kemerer [180]. Since their ap-

pearance, these OO metrics have been used to characterize, evaluate and improve the design of large applications [495]. This variety of software metrics also inspired works on new prediction models for software quality and effort estimation.

3.2.2 Early Theoretical Quality Models

The approaches for modeling software quality appeared right after the introduction of the first software metrics. One of the earliest documented quality model [574] was created by McCall et al. in 1977. McCall produced this model for the US Air Force and he attempted to bridge the gap between users and developers by focusing on a number of software quality factors that reflect both the users' views and the developers' priorities. The structure of McCall's quality model consists of three major perspectives (types of quality characteristics) for defining and identifying the quality of a software product, and each of these major perspectives consists of a number of quality factors. Each of these quality factors have a set of quality criteria, and each quality criterion could be reflected by one or more metrics. The perspectives are:

1. **Product revision**
   The product revision perspective identifies quality factors that influence the ability to change the software product, these factors are:
   - *Maintainability*, the ability to find and fix a defect.
   - *Flexibility*, the ability to make changes required as dictated by the business.
   - *Testability*, the ability to validate the software requirements.
2. **Product transition**
   The product transition perspective identifies quality factors that influence the ability to adapt the software to new environments:
   - *Portability*, the ability to transfer the software from one environment to another.
   - *Reusability*, the ease of using existing software components in a different context.
   - *Interoperability*, the extent, or ease, to which software components work together.
3. **Product operations**
   The product operations perspective identifies quality factors that influence the extent to which the software fulfills its specification:
   - *Correctness*, the functionality matches the specification.
   - *Reliability*, the extent to which the system fails.
   - *Efficiency*, system resource (including CPU, disk, memory, network) usage.
   - *Integrity*, protection from unauthorized access.
   - *Usability*, ease of use.

In total, McCall identified 11 quality factors broken down by 3 perspectives, as listed above.

In 1978, Boehm et al. [122] also defined a hierarchical model of software quality characteristics, trying to qualitatively define software quality as a set of attributes and metrics. It consists of high-level characteristics, intermediate-level characteristics and lowest level (primitive) characteristics which contribute to the overall quality level. At the highest level of his model, Boehm defined three primary uses (or basic software requirements), which are the following:

- **As-is utility**, the extent to which the as-is software can be used (i. e., ease of use, reliability and efficiency).
- **Maintainability**, ease of identifying what needs to be changed as well as ease of modification and retesting.
- **Portability**, ease of changing software to accommodate a new environment.

In the intermediate level, there are seven quality characteristics that represent the qualities expected from a software system:

- *Portability*, the extent to which the software will work under different computer configurations (i. e., operating systems, databases etc.).
- *Reliability*, the extent to which the software performs as required, i. e., the absence of defects.
- *Efficiency*, optimum use of system resources during correct execution.
- *Usability*, ease of use.
- *Testability*, ease of validation, that the software meets the requirements.
- *Understandability*, the extent to which the software is easily comprehended with regard to purpose and structure.
- *Flexibility*, the ease of changing the software to meet revised requirements.

The primitive characteristics can be used to provide the foundation for defining quality characteristics; this use is one of the most important goals established by Boehm when he constructed his quality model.

In 1995, Dromey [262] presented a product based quality model that recognizes that quality evaluation differs for each product. He realized that a more dynamic idea for modeling the evaluation process is needed to be general enough to be successfully applied for different systems. Dromey was focusing on the relationship between the quality attributes and the sub-attributes, as well as attempting to connect software product properties with software quality attributes. Dromey's quality model is structured around a 5 step process:

1. Choose a set of high-level quality attributes necessary for the evaluation.
2. List components/modules in your system.
3. Identify quality-carrying properties for the components/modules (qualities of the components that have the biggest impact on the product properties from the list).
4. Determine how each property affects the quality attributes.
5. Evaluate the model and identify weaknesses.

The FURPS [344] model was originally presented by Robert Grady at Hewlett Packard in 1992, then it has been extended by IBM Rational Software into FURPS+, where the '+' indicates such requirements as design constraints, implementation requirements, interface requirements and physical requirements. Under the FURPS model, the following characteristics are used:

- **Functionality** - it may include feature sets, capabilities, and security.
- **Usability** - it may include human factors, aesthetics, consistency in the user interface, online and context sensitive help, wizards and agents, user documentation, and training materials.
- **Reliability** - it may include frequency and severity of failure, recoverability, predictability, accuracy, and mean time between failures.
- **Performance** - it imposes conditions on functional requirements such as speed, efficiency, availability, accuracy, throughput, response time, recovery time, and resource usage.
- **Supportability** - it may include testability, extensibility, adaptability, maintainability, compatibility, configurability, serviceability, installability, and localizability.

ISO/IEC 9126 [422] is an international standard for the evaluation of software products. The standard is divided into four parts which address, respectively, the following subjects: quality model; external metrics; internal metrics; and quality in use metrics. ISO/IEC 9126 Part one, referred to as ISO/IEC 9126-1 is an extension of the work done by McCall, Boehm, Grady and others in defining a set of software quality characteristics. The standard defines six high-level product quality characteristics which are widely accepted both by industrial experts and academic researchers. These characteristics are: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. Table 3.1 shows the characteristics defined by the standard together with their sub-characteristics.

In the context of software evolution, maintainability is one of the most observed and evaluated quality characteristics (see Table 3.3). The importance of maintainability lies in its direct connection with the costs of changing the software, either by performing bug-fixes, refactoring it or adding new features. Although the source code quality directly affects maintainability, the standard does not provide a common set of source code measures as internal quality properties. The standard also does not specify how the aggregation of quality attributes should be performed. These are not deficiencies of the standard, but it offers a kind of freedom to adapt the model to specific needs.

The successor of the ISO/IEC 9126 standard family is the ISO/IEC 25000 (Systems and software Quality Requirements and Evaluation – SQuaRE) [425] family. It introduces slight modifications to the previous standard which are mainly terminology changes. Table 3.2 lists the quality characteristics and subcharacteristics of the most recent standard.

Table 3.3 provides an overview of the described theoretical quality models. The first four rows show some basic characteristics of the models based on the work of Fahmy et al. [282]. The first row contains the number of levels of the hierarchical

Table 3.1: The ISO/IEC 9126 characteristics and subcharacteristics

Characteristics	**Subcharacteristics**	**Characteristics**	**Subcharacteristics**
Functionality	Suitability Accuracy Interoperability Security Functionality Compliance	Maintainability	Analyzability Changeability Stability Testability Maintainability Compliance
Reliability	Maturity Fault Tolerance Recoverability Reliability Compliance	Efficiency	Time Behavior Resource Utilization Efficiency Compliance
Usability	Understandability Learnability Operability Attractiveness Usability Compliance	Portability	Adaptability Installability Co-Existence Replaceability Portability Compliance

Table 3.2: The ISO/IEC 25000 (SQuaRE) characteristics and subcharacteristics

Characteristics	**Subcharacteristics**	**Characteristics**	**Subcharacteristics**
Functional suitability	Functional Appropriateness Functional Correctness Functional Completeness	Portability	Adaptability Installability Replaceability
Security	Confidentalility Integrity Non-repudiation Accountability Authenticity	Usability	Appropriateness Recognisability Learnability Operability User error protection User interface aesthetics Accessibility
Maintainability	Modularity Reusability Analysability Modifiability Testability	Reliability	Availability Fault tolerance Recoverability Maturity
Performance efficiency	Time-bahaviour Resource utilisation Capability	Compatibility	Co-existence Interoperability

model. Second row shows the relation types between the quality attributes in the model. After that rows three and four highlight the main advantages and disadvantages of the particular models. In the following rows the quality attributes of the different models are presented. Only quality attributes at the highest level are considered. It can be seen that there are a lot of common properties among the models. However, only Reliability appears at the highest level in each model. Maintainability, Efficiency, Usability and Portability are also very common attributes, they

Table 3.3: Comparison of theoretical quality models [13, 282]

Characteristics	**McCall**	**Boehm**	**Dromey**	**FURPS**	**ISO 9126**	**ISO 25000**
Nr. of levels	2	3	2	2	3	3
Relationship	Many-Many	Many-Many	One-Many	One-Many	One-Many	One-Many
Main advantage	Evaluation Criteria	Hardware Factors Included	Different Systems	Separation of FR & NFR	Evaluation Criteria	Evaluation Criteria
Main disadvantage	Components Overlapping	Lack of Criteria	Comprehen-siveness	Portability not Considered	Generality	Generality
Quality Attributes						
Maintainability	✓	✓	✓		✓	✓
Flexibility	✓					
Testability	✓	✓				
Correctness	✓					
Efficiency	✓	✓	✓		✓	✓
Reliability	✓	✓	✓	✓	✓	✓
Integrity	✓					
Usability[1]	✓	✓	✓	✓	✓	✓
Portability	✓	✓	✓		✓	✓
Reusability	✓		✓			
Interoperability	✓					
Understandability		✓				
Modifiability		✓				
Functionality			✓	✓	✓	✓
Performance				✓		✓
Supportability				✓		
Security						✓
Compatibility						✓

[1] Also referred to as Human Engineering in some models

appear in 5 out of the 6 models. Moreover, these properties are contained in each model just not at the highest level everywhere (e. g., in FURPS Maintainability is included in the Supportability characteristic). On the other hand, there are also attributes that are specific to one model e. g., Supportability, Security, Compatibility, etc. Further reading about these theoretical quality models can be found in the work of Al-Qutaish [13].

3.2.3 Metrics-based Empirical Prediction Models

The first step towards applying quality models in practice was the development of different empirical models. All these models apply software metrics as quality predictors.

One of the most widely known empirical maintainability prediction models is the *Maintainability Index* (MI) [662] introduced in 1997 by the Carnegie Mellon Software Engineering Institute (SEI). The formula has many derivatives, but the original form is given in Equation 3.1. A common variation, shown in Equation 3.2, adds the comment lines to the model.

$$MI = 171 - 5.2 * \ln V - 0.23 * G - 16.2 * \ln LOC \tag{3.1}$$

$$MI = 171 - 5.2 * \log_2 V - 0.23 * G - 16.2 * \log_2 LOC + 50 * \sin(\sqrt{2.46 * CM}) \tag{3.2}$$

The applied measures are the following:

- V - Halstead Volume.
- G - Cyclomatic Complexity.
- LOC - count of source Lines Of Code (SLOC).
- CM - percent of lines of Comments.

The CM percentage in the maintainability index formula has been interpreted in two different ways. Liso [530] assumed CM to range between 0 and 100 and discussed the appropriateness of the value 2.46 leading to strange peaks in $\sin(\sqrt{2.46 * CM})$. Thomas [831] has assumed CM to range between 0 and 1.

Later on, many variations of the formula have been introduced, e. g., one of its derivatives is built into Microsoft Visual Studio as well. However, the effectiveness and usefulness of the maintainability index is and has been a subject of debate [142, 289, 387, 483].

Therefore, a wide variety of new approaches has been introduced for improving the MI. The applied methods are ranging from regression models to fuzzy aggregation and Bayes classifiers (see Table 3.4). These empirical studies also differ in which metrics have found to be the most effective maintainability predictors. Riaz et al. presented a detailed comparative study [714] on existing empirical maintainability prediction models. The work collects many important features of the different empirical models, e. g., the definition of quality the authors used, the applied validation methodology, or the accuracy of the model. Table 3.4 contains a summary of some well-known works in the field and their important properties.

Table 3.4: A summary of empirical maintainability prediction models [714]

Authors	**Year**	**Approach**	**Maintainability Metrics**
Oman, and Hagemeister [663]	1994	3 regression based models: 1. Single metric model based on Halstead's Effort 2. A four-metric polynomial model 3. A five-metric linear regression model	Subjective assessment (ordinal scale metric) by using the US Air Force Operational Test and Evaluation Center's software maintainability evaluation instrument, which provides a rating as well as categorizes maintainability as low, medium or high
Coleman, Ash, Lowther, and Oman. [196, 197]	1994 1995	1. HPMAS (Hewlett Packard's software Maintainability Assessment System) 2. Polynomial maintainability assessment model	Same as first but they call it HPMAS Maintainability Index
Welker, Oman [912]	1997	1. Improved, three-metric MI model 2. Improved, four-metric MI model	Same as first but they call it HPMAS Maintainability Index
Genero, Olivas, Piattini, and Romero, F. [319]	2001	Fuzzy Prototypical Knowledge Discovery used for prediction based on Fuzzy Deformable Prototypes	Expert opinion using an ordinal scale
Misra [609]	2005	6 models based on multivariate regression analysis	Maintainability Index (MI)
Van Koten, Gray [869]	2006	1. Bayesian Network - Naive-Bayes Classifier 2. Regression Models (Regression Tree, Multiple linear regression models)	CHANGE metric: count of LOC changed during a 3-year maintenance period
Shibata, Rinsaka, Dohi, and Okamura [774]	2007	3 Non-Homogeneous Poisson Process based Software Reliability Models (NHPP-based SRMs): 1. Exponential SRM 2. S-Shaped SRM 3. Rayleigh SRM	Their own queueing model with an infinite number of servers, which is related to the software fault-detection/correction profiles
Zhou, and Leung [952]	2007	Multivariate Linear Regression, Artificial Neural Network, Regression Tree, Support Vector Regression, Multivariate Adaptive Regression Splines	CHANGE metric: count of LOC changed during a 3-year maintenance period
Zhou, and Xu [953]	2008	1. Univariate Linear Regression Analysis 2. Multivariate Linear Regression Model	Maintainability Index (MI)

3.2.4 State-of-the-art Practical Quality Models

The appearance of the widely accepted ISO/IEC 9126 and related standards [422] has pushed forward the research in the field of quality models. Numerous papers, ranging from highly theoretical to purely practical ones, are dealing with this important research area. Some of the research has focused on developing a methodology for adapting the ISO/IEC 9126 model in practice [119, 807]. They provide guidelines or a framework for constructing effective quality models.

This section focuses more on practical models that are directly applicable for assessing the quality of software systems. Using the results of static source code analysis is one of the most widespread solutions to calculate an external quality attribute from internal quality attributes [71]. There are several case studies examining if metrics are appropriate indicators for external quality attributes such as code fault proneness [357, 660], maintainability [59] and attractiveness of the user interface [584].

The majority of these practical models consider the maintainability aspect of quality only, because it is the easiest characteristic to assess based on pure source code analysis. Some of the models consider other quality attributes as well, like usability (often requiring manual input for the qualification). Regarding the terminology, we use quality model and maintainability model as synonyms throughout this section.

3.2.4.1 Software QUALity Enhancement project (SQUALE)

The SQUALE model presented by Mordal et al. [619] introduces so-called practices to connect the ISO/IEC 9126 characteristics to metrics. A practice in a source code element expresses a low-level rule and the reparation cost of violating this rule. The reparation cost of a source code element is calculated by the sum of the reparation costs of its rule violations. The practices can use multiple source code measures like complexity, lines of code, coding rule violations, etc. (e. g., the comment rate practice uses the measures cyclomatic complexity v(G) and source code lines, SLOC). Based on the measures, a practice rating in the [0;3] interval can be calculated, where 3 means the fully achieved goal, 0 means not achieved goal, 1 and 2 means partly achieved goal. In the case of the comment rate practice, the rating can be determined according to the following rule:

Comment rate practice

if $v(G) < 5$ *and* $SLOC < 30$ **then**
  $rating = 3$
else
  $rating = \frac{\%\_comments\_per\_loc}{1-10^{(-v(G)/15)}}$
end if

A criterion assesses one principle of software quality (e. g., safety, simplicity, or modularity) and it aggregates a set of practices. A criterion mark is computed as the weighted average of the composed practice marks. There are different weighting profiles, e. g., hard, medium, soft.

A factor represents the highest quality assessment to provide an overview of project health (e. g., functional capacity or reliability). A factor aggregates a set of criteria and its mark is computed as the average of the composed criteria marks.

The model also defines a so-called improvement plan that gives the order in which the elements should be improved. The plan is based on how to achieve the biggest improvement in the rating with the lowest invested effort.

3.2.4.2 Software Quality Assessment based on Lifecycle Expectations (SQALE)

The SQALE quality model introduced by Letouzey and Coq [516] is basically a requirements model. Assessing software source code is therefore similar to measuring the distance which separates it from its quality target.

The model consists of quality characteristics built on top of development activities following one another. The characteristics are taken from the ISO/IEC 9126 standard; however, they are grouped differently and their subcharacteristics are changed entirely. Each subcharacteristic is measured by a number of different control points. The control points are base measures (indicators) that measure different non-compliance aspects of the source code. e. g., an understandability (a subcharacteristic of maintainability) indicator is the file comment ratio. If it is below SQALE's default threshold of 25%, a violation is counted.

Every rule violation has a remediation effort (which depends on the rule). The model calculates an index for every characteristic which is the sum of all the remediation efforts of its rule violations. The index represents the remediation effort which would be necessary to correct the non-compliances detected in the component, versus the model requirements. Since the remediation index represents a work effort, the consolidation of the indices is a simple addition of uniform information. In this way coding rule violation non-compliances, threshold violations for a metric or the presence of an antipattern non-compliance can be compared using their relative impact on the index.

Besides these remediation indices the model presents a five level rating for the different components or the system as a whole. The ratings are A, B, C, D, E (A being the best, E the worst) and can be calculated by summing the remediation costs of the rule violations for a component divided by the average development cost of reimplementing the same component (estimated from LOC). Based on preset thresholds for this ratio a rating can be derived (e. g., if the ratio is less than 0.1% then the rating is A).

3.2.4.3 Quamoco Quality Model

The Quamoco quality framework [905] is the result of a German national research project carried out between 2009 and 2011. The Quamoco Consortium – consisting of research institutions and companies – has developed a quality standard applicable in practice that makes the performance and efficiency of software products made in Germany assessable and accountable.

Quamoco is based on practical experiences learnt from existing quality models. The high-level of detail of this approach for the qualified certification of software projects also takes into account the diversity of different software products. This means that Quamoco contains a basic standard of quality that is complemented by domain-specific quality standards. The quality of software products can thus be modeled flexibly. At the same time, Quamoco ensures that all identified quality requirements are fully integrated.

The Quamoco approach uses the following definitions:

- Quality Model: A model with the objective to describe, assess and/or predict quality.
- Quality Meta Model: A model of the constructs and rules needed to build specific quality models.
- Quality Modeling Framework: A framework to define, evaluate and improve quality. This usually includes a quality metamodel as well as a methodology that describes how to instantiate the metamodel and use the model instances for defining, assessing, predicting and improving quality.

The main concepts of the quality model are *Factors*. A factor expresses a property of an entity. Entities are the things that are important for quality. Properties describe the attributes of the entities. This concept of a factor is rather general. Thus, the Quamoco model uses it on two levels of abstraction:

- Quality Aspects describe abstract quality goals defined for the whole product. The quality model uses the "-ilities" of ISO/IEC 25000 as quality aspects. Typical examples for such quality aspects are Maintainability, Analysability, and Modifiability.
- Product Factors describe concrete, measurable properties of concrete entities. An example for a factor is the Complexity of a method, which can be measured by the cyclomatic complexity number, or by the nesting depth of the method.

To close the gap between abstract quality aspects and measurable product factors, the product factors need to be set in relation to the quality aspects. This is done via *Impacts*. An impact is either positive or negative and describes how the degree of presence or absence of a product factor influences a quality aspect.

A third layer in the levels of abstraction are *Measures*, which describe how a specific product factor can be quantified. To realize the connection to concrete tools in a quality assessment, the approach further introduces *Instruments*. An instrument describes a concrete implementation of a measure. For the example of the nesting

depth, an instrument is the corresponding metric as implemented in the quality analysis framework ConQAT [234]. This way, different tools can be used for a single measure.

In order to fully utilize the quality model, aggregation formulas need to be specified. They are called *Evaluations* and they are assigned to the factors in the quality model.

3.2.4.4 SIG Maintainability Model

Kuipers and Visser introduced a maintainability model [483] as a replacement of the Maintainability Index by Oman and Hagemeister [662]. Based on this work Heitlager et al. [387], members of the Software Improvement Group (SIG) company proposed an extension of the ISO/IEC 9126 model that uses source code metrics at low-level. Metric values are split into five categories, from poor (−−) to excellent (++). The evaluation in their model means summing the values for each attribute (having the values between -2 and +2) and then aggregating the values for characteristics using the mapping presented in Table 3.5. The model was recently adapted to the ISO/IEC 25000 standard.

Table 3.5: The SIG quality characteristic mapping

	Volume	**Complexity**	**Duplications**	**Unit size**	**Unit tests**
Analysability	✓		✓	✓	✓
Changeability		✓	✓		
Stability					✓
Testability		✓		✓	✓

Correia and Visser [202] presented a benchmark that collects measurements of a wide selection of systems. This benchmark enables systematic comparison of technical quality of (groups of) software products. Alves et al. presented a technique for deriving metric thresholds from benchmark data [24]. This method is used to derive more reasonable thresholds for the SIG model as well.

Correia and Visser [203] introduced a certification method that is based on the SIG quality model. The method makes it possible to certify technical quality of software systems. Each system can get a rating from one to five stars (−− corresponds to one star, ++ to five stars). Baggen et al. [58] refined this certification process by doing regular re-calibration of the thresholds based on the benchmark.

The SIG model uses binary relation between system properties and characteristics. Correia et al. created a survey [201] to elicit weights for their model. The survey was filled out by IT professionals, but the authors finally concluded that using weights does not improve their quality model because of the lack of consensus among developers.

The validation of the model has been done through an empirical case study. Luijten and Visser [543] showed that the metrics of the SIG quality model correlate with the time needed for resolving a defect in a software.

3.2.4.5 Columbus Quality Model

This subsection describes the Columbus Quality Model (ColumbusQM) in full technical details to give an insight for the reader about the complexity of a modern maintainability model.

The Columbus approach [63] to compute ISO/IEC 9126 quality characteristics uses a so-called benchmark (i. e., a source code metric repository database consisting of source code metrics of open-source and industrial software systems) and it is based on a directed acyclic graph (see Figure 3.2), whose nodes correspond to quality properties that can either be internal or external. The nodes representing internal quality properties are called *sensor nodes* (white nodes in Figure 3.2) as they measure internal quality directly. The other nodes are called *aggregate nodes* as they acquire their measures through aggregation. The approach uses aggregate nodes defined by the ISO/IEC 9126 standard (dark gray nodes in Figure 3.2) as well as newly defined ones (light gray nodes in Figure 3.2).

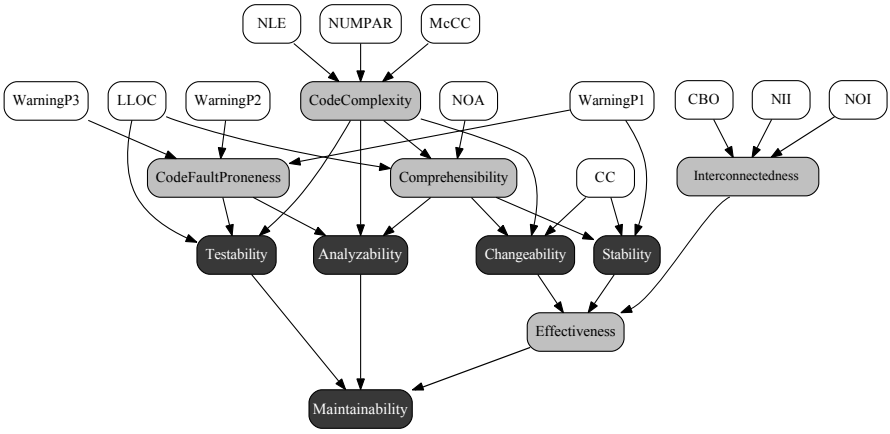

Fig. 3.2: Java Attribute Dependency Graph of ColumbusQM

The edges of the graph represent dependencies between an internal and an external or two external properties. Internal properties are not dependent on any other attribute, they "sense" internal quality directly. The aim is to evaluate all the external quality properties (attributes) by performing an aggregation along the edges of the graph. In the following we will refer to this graph as *Attribute Dependency Graph (ADG)*.

Let $G = (S \cup A, E)$ stand for the *ADG*, where S, A, and E denote the sensor nodes, aggregate nodes, and edges, respectively, and $S \cap A = \emptyset$. We want to measure how good or bad an attribute is. *Goodness* is the term that is used to express this measure of an attribute. For the sake of simplicity we will write goodness of a node instead of goodness of an attribute represented by a node. Goodness is measured on the $[0,1]$ interval for each node, where 0 and 1 mean the worst and best, respectively. The goodness of each sensor node u is not known precisely, hence it is represented by a random variable X_u with a probability density function $g_u : [0,1] \to \mathbb{R}$. g_u is called the *goodness function* of node u.

Constructing a goodness function. The currently presented way of constructing goodness functions is specific to source code metrics. For other sensor types, different approaches may be needed. The model makes use of the metric histogram over the source code elements, as it characterizes the whole system from the aspect of one metric. The aim is to give a measure for the goodness of a histogram. As the notion of goodness is relative, it is expected to be measured by means of comparison with other histograms in the benchmark. Let us suppose that H_1 and H_2 are the histograms of two systems for the same metric, and $h_1(t)$ and $h_2(t)$ are the corresponding normalized histograms (i. e., density functions, see Figure 3.3). By using Equation 3.3 we obtain a distance function (not in the mathematical sense) defined on the set of probability functions. Fig. 3.3 helps understanding the meaning of the formula: it computes the signed area between the two functions weighted by the function $\omega(t)$.

$$\mathscr{D}(h_1, h_2) = \int_{-\infty}^{\infty} (h_1(t) - h_2(t))\, \omega(t)\, dt \tag{3.3}$$

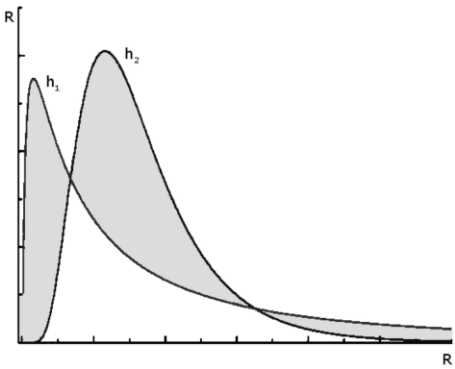

Fig. 3.3: Comparison of probability density functions

The weight-function plays a crucial role: it determines the notion of goodness, i. e., where on the horizontal axis the differences matter more. If one wants to express that all metric values matter in the same amount, she would set $\omega(t) = c$, where c is a constant, and in that case $\mathscr{D}(h_1, h_2)$ will be zero (as h_1 and h_2 inte-

grate to 1). On the other hand, if one would like to express that higher metric values are worse, one could set $\omega(t) = t$. Non-linear functions for $\omega(t)$ are also possible. As in case of most source code metrics, higher values are considered to be worse (e. g., McCabe's complexity), we use the $\omega(t) = t$ weight function for these metrics (linearity is implicitly subsumed by the choice).

The choice leads to a very simple formula, given in Equation 3.4, where H_1' and H_2' are the random variables corresponding to the h_1 and h_2 density functions, $E\left(H_1'\right)$ and $E\left(H_2'\right)$ are the expected values of these (the equality is based on the definition of the expected value of a random variable). Lastly, $\tilde{H}_1$ and $\tilde{H}_2$ are the averages of the histograms H_1 and H_2, respectively. The last approximation is based on the Law of Large Numbers (the averages of a sample of a random variable converge to the expected value of the same). By this comparison we get one goodness value for the subject histogram (this value is relative to the other histogram).

$$\begin{aligned}\mathscr{D}(h_1,h_2) &= \int_{-\infty}^{\infty}(h_1(t)-h_2(t))\,tdt = \int_{-\infty}^{\infty}h_1(t)\,tdt - \int_{-\infty}^{\infty}h_2(t)\,tdt \\ &= E\left(H_1'\right) - E\left(H_2'\right) \approx \tilde{H}_1 - \tilde{H}_2\end{aligned} \tag{3.4}$$

In order to obtain a proper goodness function, this comparison needs to be repeated with histograms of many different systems independently. In each case we get a goodness value which can basically be regarded as sample of a random variable from the range $[-\infty,\infty]$. A linear transformation of the values changes the range to the $[0,1]$ interval. The transformed sample is considered to be the sample of the random variable X_u. Interpolation of the empirical density function leads to the goodness function of the sensor node.

There is a theoretical beauty of the approach. Let us assume that one disposes histograms of N different systems for one particular metric. Each histogram can be considered to be sampled by different random variables $Y_i, (i = 1,\ldots,N)$. Furthermore, one would like to assess the goodness of another histogram corresponding to the random variable X. The goodness is by definition described by the series of random variables in Equation 3.5. The random variable for goodness (before the transformation) is then described by random variable Z in Equation 3.6.

$$Z_1 := E(Y_1) - E(X), \ldots, Z_N := E(Y_N) - E(X). \tag{3.5}$$

$$Z := \frac{1}{N}\sum_{i=1}^{N} Z_i \to \Phi_{\nu,\sigma}, \text{ if } N \to \infty. \tag{3.6}$$

According to the Central Limit Theorem [266] for independent (not necessarily identically distributed) random variables, Z tends to a normal distribution which is independent of the benchmark histograms. This is naturally a theoretical result, and it states that when having a large number of systems in the benchmark, the constructed goodness functions are (almost) independent of the particular systems in the benchmark. Actually, $\Phi_{\nu,\sigma}$ is a benchmark-independent goodness function

(on $[-\infty,\infty]$) for X, just that it can be approximated by having a benchmark with a sufficient number of systems.

To be able to perform the construction of goodness functions in practice, a source code metric repository database has also been built that consists of source code metrics of more than 100 open-source and industrial software systems.

Aggregation. After being able to construct goodness functions for sensor nodes, there is a need for a way to aggregate them along the edges of the ADG. Recall that the edges represent only dependencies, we have not yet assigned any weights to them. Assigning a simple weight would lead to the classic approach. In models that use a single weight or threshold in aggregation, the particular values are usually backed up with various reasonings and cause debates among experts. The Columbus model is able to handle this ambiguity. Many experts were asked in an online survey (both industrial and academic people) for their opinion about the weights. For every aggregate node, they were asked to assign scalars to incoming edges such that the sum of these would be 1. The number assigned to an edge is considered to be the amount of contribution of source goodness to target goodness. This way, for each aggregate node v a multi-dimensional random variable $\mathbf{Y}_v = \left(Y_v^1, Y_v^2, \ldots, Y_v^n\right)$ exists (n is the number of incoming edges). The components are dependent random variables, as

$$\sum_{i=1}^{n} Y_v^i = 1, \tag{3.7}$$

holds, that is, the range of $\mathbf{Y}_v$ is the standard $(n-1)$-simplex in $\mathbb{R}^n$. It is important that one cannot simply decompose $\mathbf{Y}_v$ to its components because of the existing dependencies among them.

Having an aggregate node with a composed random variable $\mathbf{Y}_v$ for aggregation ($\mathbf{f}_{\mathbf{Y}_v}$ will denote its composed density function), and also having n source nodes along the edges, with goodness functions $g_1, g_2, \ldots g_n$, the aggregated goodness for the aggregated node is defined by $g_v(t)$ in Equation 3.8 where Δ^{n-1} is the $(n-1)$-standard simplex in $\mathbb{R}^n$ and C^n is the standard unit n-cube in $\mathbb{R}^n$.

$$g_v(t) = \int_{\substack{t=\mathbf{qr} \\ \mathbf{q}=(q_1,\ldots,q_n)\in\Delta^{n-1} \\ \mathbf{r}=(r_1,\ldots,r_n)\in C^n}} \mathbf{f}_{\mathbf{Y}_v}(\mathbf{q})\, g_1(r_1)\ldots g_n(r_n)\, d\mathbf{r} d\mathbf{q}, \tag{3.8}$$

It is the generalization of how aggregation is performed in classic approaches. Classically, a linear combination of goodness values and weights is taken, and it is assigned to the target node. When dealing with probabilities, one needs to take every possible combination of goodness values and weights, and also the probabilities of their outcome into account. In the formula, the components of the vector $\mathbf{r}$ traverse the domains of source goodness functions independently, while vector $\mathbf{q}$ traverses the simplex where each point represents a probable vote for the weights. For fixed $\mathbf{r}$ and $\mathbf{q}$ vectors their scalar product ($t = \mathbf{qr} = \sum_{i=1}^{n} r_i q_i \in [0,1]$) is the goodness of the target node. To compute the probability for this particular goodness value, one needs to multiply the probabilities of goodness values of source nodes (these are independent) and also the composed probability of the vote ($\mathbf{f}_{\mathbf{Y}_v}(\mathbf{q})$). This product is integrated over all the possible $\mathbf{r}$ and $\mathbf{q}$ vectors (please note that t is not uniquely

decomposed to vectors $\mathbf{r}$ and $\mathbf{q}$). $g_v(t)$ is indeed a probability distribution function on $[0,1]$ interval, i. e., its integral is equal to 1, because both $\mathbf{f}_{\mathbf{Y}_v}(\mathbf{q})$ and the goodness functions integrate to 1 on Δ^{n-1} and C^n respectively.

With this method it is now possible to compute goodness functions for every aggregate node. The way the aggregation is performed is mathematically correct, meaning that the goodness functions of aggregate nodes are really expressing the probabilities of their goodness (by combining other goodness functions with weight probabilities).

Although this approach provides goodness functions for every aggregate node, managers are usually only interested in having one number that represents an external quality attribute of the software. Goodness functions carry much more information than that, but an average of the function may satisfy even the managers. The resulting goodness function at every node has a meaning: it is the probability distribution which describes how good a system is from the aspect represented by the node. Therefore, the approach leads to *interpretable* results. Provided that the goodness functions are computed for every node of the *ADG*, and that the dependencies in the *ADG* are known, it is easy to see the root causes of the quality score.

Drill-down. Additionally to system level maintainability, ColumbusQM implements an algorithm [379] to drill down to lower levels in the source code and to get a similar measure for the building blocks of the codebase (e. g., classes or methods). For this, the model defines the **relative maintainability index** for the source code elements, which measures the extent to which they affect the system level goodness values. The approach is related to the aggregation and decomposition techniques introduced by Posnett et al. [699] and Serebrenik et al. [769].

The basic idea is to calculate the system level goodness values by ColumbusQM, leaving out the source code elements one by one. After a particular source code element is left out, the system level goodness values will change slightly for each node in the ADG. The difference between the original goodness value computed for the system, and the goodness value computed without the particular source code element, will be called the *relative maintainability index* of the source code element itself. The relative maintainability index is a small number that is either positive when it improves the overall rating or negative when it decreases the system level maintainability. The absolute value of the index measures the extent of the influence to the overall system level maintainability. A relative index can be computed for each node of the ADG, meaning that source code elements can affect various quality aspects in different ways and to different extents.

It is important to notice that this measure determines an ordering among the source code elements of the system, i. e., they become comparable to each other. And what is more, the system level maintainability being an absolute measure of maintainability, the relative index values become absolute measures of all the source code elements in the benchmark. Therefore, the ordering can be used by programmers to rank source code elements based on their criticality for improving the overall maintainability.

3.2.4.6 Other Approaches

The CAST (http://www.castsoftware.com) company has its own solution for software quality analysis, the Application Intelligence Platform (AIP), that uses an application quality benchmarking repository called Appmarq. Being a closed source proprietary tool, we were unable to try it out and evaluate it in detail as we did this with other solutions.

The Laboratory for Quality Software (LaQuSo – http://www.laquso.com) is a joint initiative of Eindhoven University of Technology and Radboud University Nijmegen. Since the starting of LaQuSo one of its focus areas has been the development of a product certification methodology. This has resulted in LSPCM (LaQuSo Software Product Certification Model). LaQuSo offers product certification as a service, which is a check that the artifact fulfills a well-defined set of requirements. These requirements are defined by the customer or a third party; LaQuSo as an independent evaluator will do the check. Serebrenik et al. [766] have analyzed requirements of three off-shoring projects using LSPCM. Application of LSPCM revealed severe flaws in one of the projects. The responsible project leader confirmed later that the development significantly exceeded time and budget. In the other projects no major flaws were detected by LSPCM and it was confirmed that the implementation was delivered within time and budget.

VizzMaintenance (http://arisa.se/products.php) is an Eclipse plug-in which brings detailed information about the maintainability of a software system. It uses static analysis to calculate 17 well-known software quality metrics. It then combines these values in a software quality model [928]. It supports the decisions which classes should be refactored first to improve their maintainability.

Vanderose et al. introduce a Model-Centric Quality Assessment (MoCQA) framework [593, 875, 877] which is a theoretical framework designed to help plan and support a focused quality assessment all along the software lifecycle. They aim at assessing other quality characteristics than maintainability, such as completeness [876], that are arguably useful to assist the software maintenance process.

There are other works that deal with software design quality and quality from the end user's point of view. For example, Ozkaya et al. [674] emphasize the importance of using quality models like ISO/IEC 9126 in practice right from the beginning of the design phase. The approach presented in their paper is general enough for evaluating design or end user quality, but not the product quality itself. Research of Bansiya and Davis [68] focus on the software design phase. They adapted the ISO/IEC 9126 model for supporting quality assessment of system design.

The work of Marinescu and Lanza [495] introduces a metrics-based approach for detecting design problems. It allows the software engineer to define metrics-based rules that "quantify" design principles, rules and heuristics related to the quality of a design. The work introduces an important suite of detection strategies for the identification of different well-known design flaws found in the literature. Additionally, the work presents a new type of quality model, called Factor-Strategy, allowing the quality to be expressed explicitly in terms of compliance with principles, rules and heuristics of good object-oriented design.

3.3 Application of Practical Quality Models in Software Evolution

It might be difficult to see the role of the presented maintainability models in software evolution at first glance. But whether one likes it or not, today, software industry is a giant business driven by business needs and profit. Thus keeping the costs of software evolution as low as possible is a central issue. As maintainability is in direct connection with the changing of software systems, measuring and controlling it is of vital importance for software evolution.

On the other hand, as applying techniques that improve the maintainability of the code or avoiding structures that deteriorate systems has an additional cost without having a short term financial benefit, they are often neglected by the business stakeholders. Hence maintainability of the systems is often overshadowed by feature developments whose business value is more evident at least in short terms. Although the developers are usually aware of its long term benefits, they do not have strong enough arguments to convince stakeholders for investing extra effort to improve maintainability. By better understanding the relationship of maintainability and the long term development costs, it would be possible to show the return on investment of keeping maintainability of systems at a high-level. It would make the extra investment more appealing to the business stakeholders as well, thus reaching higher quality software and cheaper evolution in general.

In this section we introduce a cost model that is based on source code maintainability and proves its direct connection with development costs. It is a possible application of practical quality models during software evolution in modern industrial environments.

3.3.1 A Cost Model Based on Software Maintainability

The approach [62] adopts the concept of entropy in thermodynamics, which is used to measure the disorder of a system. According to the second law of thermodynamics, the entropy of a closed system cannot be reduced; it can only remain unchanged or increase. The only way to decrease entropy (disorder) of a system is to apply external forces, i.e. to put energy into making order.

The notion of entropy is applied in a very similar way for software systems [432]. Maintainability of a source code is usually defined as a measure of the effort required to perform specific modifications in it. Assuming that the higher the disorder is, the more effort is needed to perform the modifications, maintainability can be interpreted as a measure of the disorder, i.e. entropy of the source code.

The approach lays on two basic assumptions:

1. Making changes in a source code does not decrease the disorder of it, provided that one does not work actively against this. In other words, when making

changes to a software system without explicitly aiming to improve it, its maintainability will decrease, or at least it will remain unchanged.
2. The amount of changes applied to the source code is proportional to the effort invested, and to the maintainability of the code. In other words, if one applies more effort, the code will change faster. Additionally, a more maintainable code will change faster, even if the applied effort is the same. Another interpretation is that the effort aiming on code change is inversely proportional to the maintainability at time t.

Before formalizing these assumptions, the following notions are introduced:

- $\mathscr{S}(t)$ - the size of the source code at time t, measured in lines of code.
- $\lambda(t)$ - the change rate of the source code at time t, i.e. the probability of changing any line independently (for the sake of simplicity we assume that it is the same for all lines of code). $\mathscr{S}(t)\lambda(t)$ equals the number of lines changed at time t.
- k - a constant for the conversion between different units of measure. The approach deals with two scalar measures: maintainability and cost. Instead of fix particular units of measure for each, a conversion constant k is introduced. In the sequel, it can be assumed without the loss of generality, that cost is expressed by any measure of effort, e.g. salary, person month, time, etc., while maintainability may have any other scalar measure. In practice, after fixing the measures of unit for each, k can be estimated from historical project data.
- $\mathscr{C}(t)$ - the cost invested into changing the system until time t, measured from an initial time $t = 0$. Obviously, $\mathscr{C}(0) = 0$.
- $\mathscr{M}(t)$ - maintainability (i.e. disorder) of the system at time t.

In the following, it is assumed that modifications do not explicitly aim on code improvement, meaning that only new functionality is being added to the system and no refactoring or other explicit improvements are done. In this case, the first assumption above can be formalized as in Equation 3.9, meaning that the decrease rate of maintainability is proportional to the number of lines changed at time t. The constant factor q is called the *erosion factor* which represents the amount of "damage" (decrease in maintainability) caused by changing one line of code.

$$\frac{d\mathscr{M}(t)}{dt} = -q\mathscr{S}(t)\lambda(t) \qquad (q \geq 0), \tag{3.9}$$

The erosion factor depends on many internal and external factors like the experience and knowledge of the developers, maturity of development processes, quality assurance processes used, tools and development environments, the programming language, and the application domain. The $q \geq 0$ assumption makes it impossible for the code to improve by itself just by adding new functionality. The assumption is in accordance with Lehman's laws [511] of software evolution, which state that the complexity of evolving software is increasing, while its quality is decreasing at the same time.

Formalizing the second assumption leads to Equation 3.10. The numerator represents the amount of change introduced at time t. The formula states that the utilization of the cost invested at time t for changing the code is inversely proportional to maintainability.

$$\frac{d\mathscr{C}(t)}{dt} = k\frac{\mathscr{S}(t)\,\lambda(t)}{\mathscr{M}(t)} \tag{3.10}$$

Solving the above system of ordinary differential equations, yields the following result:

$$\begin{aligned}\mathscr{C}(t_1) - \mathscr{C}(t_0) &= \int_{t_0}^{t_1} k\frac{\mathscr{S}(t)\,\lambda(t)}{\mathscr{M}(t)}dt = -\frac{k}{q}\int_{t_0}^{t_1}\frac{\dot{\mathscr{M}}(t)}{\mathscr{M}(t)}dt = \\ &= -\frac{k}{q}\left[\ln\mathscr{M}(t_1) - \ln\mathscr{M}(t_0)\right] = -\frac{k}{q}\ln\frac{\mathscr{M}(t_1)}{\mathscr{M}(t_0)}.\end{aligned} \tag{3.11}$$

By expressing $\mathscr{M}(t)$ from the above equation, we get to the main result:

$$\mathscr{M}(t_1) = \mathscr{M}(t_0)\,e^{-\frac{q}{k}(\mathscr{C}(t_1)-\mathscr{C}(t_0))}, \tag{3.12}$$

which suggests that the maintainability of a system decreases exponentially with the invested cost to change the system. The erosion factor q determines the decrease rate of maintainability. It is obvious that for a higher erosion factor the decrease rate will be higher as well. It is crucial for software development companies to push the erosion factor as low as possible, for instance by training the employees, improving processes, utilizing sophisticated quality assurance technologies.

Although, the formula does not provide a way of having an absolute measure for maintainability, one can easily define a *relative maintainability* for the system. Indeed, by letting $t_0 = 0$, and defining $\mathscr{M}(0) = 1$, we get to the following function for maintainability:

$$\mathscr{M}(t) = e^{-\frac{q}{k}\mathscr{C}(t)} \tag{3.13}$$

For the interpretation, let us consider two artificial scenarios. Figure 3.4 shows the case, when the invested effort is constant over the time. In this case, both the maintainability $\mathscr{M}(t)$ and the change rate $\lambda(t)$ decrease exponentially.

In the other case, let us suppose that one intentionally wants to keep the change rate of the system constant. Figure 3.5 shows how the maintainability $\mathscr{M}(t)$ and the overall cost $\mathscr{C}(t)$ change over time. Now, the maintainability decreases linearly until it reaches zero, while the cost is increasing faster than an exponential rate. The cost will reach infinity in finite time, exactly when maintainability reaches zero, meaning that any further change would require infinite amount of effort. This is, of course, just a theoretical possibility, as no one disposes an infinite amount of resources required to degrade the maintainability of a system to absolute zero.

The problem with applying the model to real-world software systems lies in the erosion factor q. While the other model parameters (k and $\mathscr{C}(t)$) can be computed easily, the erosion factor, which measures the "damage" caused by changing one

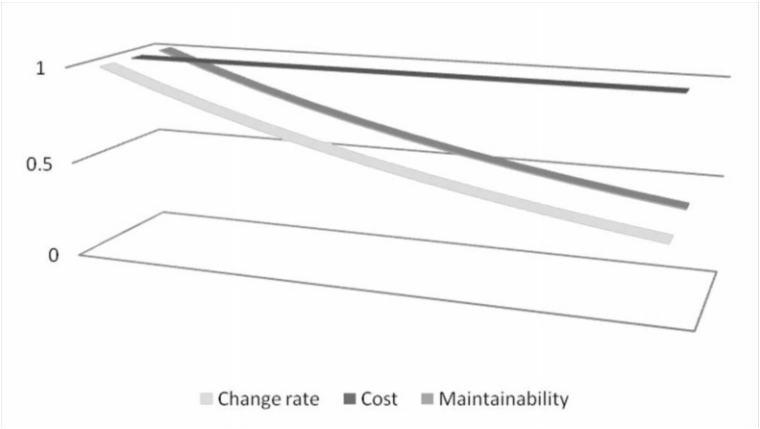

Fig. 3.4: Changes of *Change rate* ($\lambda(t)$) and *Maintainability* ($\mathscr{M}(t)$) during evolution when the cost of the development ($\mathscr{C}(t)$) is constant over time.

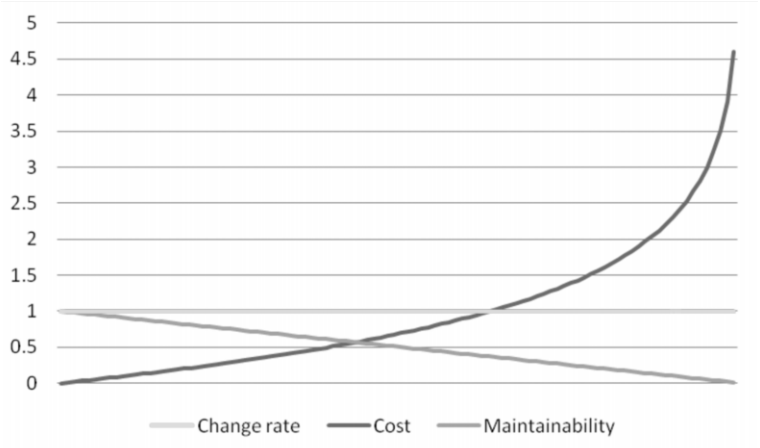

Fig. 3.5: Changes of *Cost* ($\mathscr{C}(t)$) and *Maintainability* ($\mathscr{M}(t)$) during evolution when the *Change rate* ($\lambda(t)$) is constant over time.

line, is challenging. Contrarily, if there was an absolute measure of maintainability, the constant, project-specific erosion factor q could easily be computed by expressing it from Equation 3.13. Furthermore, by having an absolute measure for q as well, the erosion factors of different projects, organizations could be compared. The analysis of the causes of the differences would make it possible to lower the erosion factor, e. g., by improving the processes, and training people.

In addition, the overall cost of development could also be expressed explicitly from the model, according to Equation 3.14. For computing future development

costs, it would just be required to have an estimate for the change rate $\lambda(t)$ over a time period.

$$\mathscr{C}(t) = -\frac{k}{q} \ln \left| 1 - \frac{q}{\mathscr{M}(0)} \int_0^t \mathscr{S}(s)\,\lambda(s)\,ds \right|. \tag{3.14}$$

The introduced practical quality models are good candidates for obtaining an absolute measure of maintainability for software systems. Using the absolute maintainability calculated by one of these models would allow to obtain an absolute erosion factor q, which can be used to estimate further development costs and to compare the erosion factors of different projects and organizations.

3.4 Tools Supporting Software Quality Estimation

3.4.1 Software QUALity Enhancement project (SQUALE)

The implementation of the SQUALE model (see Section 3.2.4.1) is available as an open-source tool (`http://www.squale.org`). The project officially started in June 2008, funded by the French Government. The first official open-source version was released in January 2009.

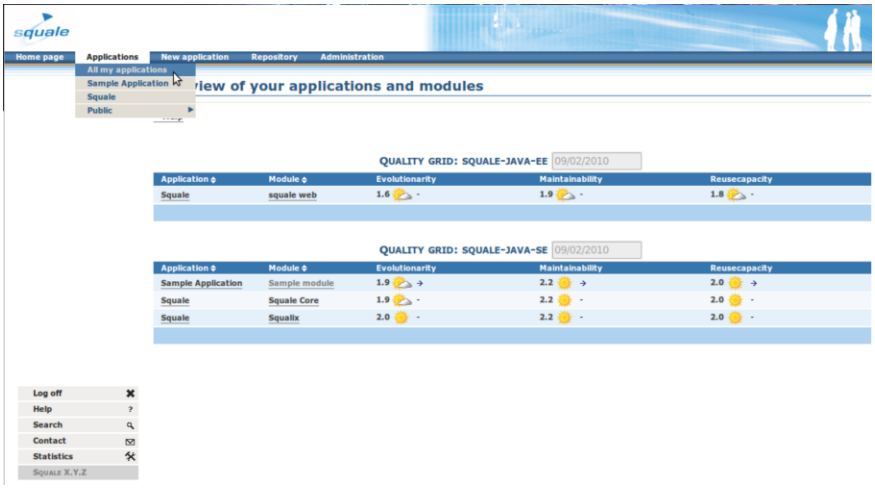

Fig. 3.6: SQUALE tool

The Software QUALity Enhancement project – SQUALE focused on two main aspects. First, it works on enhanced quality models inspired by existing approaches

(GQM [874], McCall et al. [574]) and standards (ISO/IEC 9126 [422]), validated and improved by researchers, dealing with both technical and economical aspects of quality. Second, the development of an open-source application that helps assessing software quality and improving it over time based on third party technologies (commercial or open-source) that produce raw quality information (like metrics), using the quality models to aggregate this raw information into high-level quality factors, all this targeting different languages.

The tool provides a web-based interface for configuring the qualifications of new applications. The qualification process is run as the part of a scheduled audit of the source code. The quality results are presented in the same web application. Figure 3.6 shows the overview page of a quality audit result of SQUALE.

3.4.2 *Software Quality Assessment based on Lifecycle Expectations (SQALE)*

According to the official site (http://www.sqale.org/tools), the following tools implement the SQALE model (see Section 3.2.4.2):

- Insite SaaS by Metrixware (http://www.metrixware.com)
- Sonar by SonarSource (http://www.sonarsource.com)
- SQuORE by SQuORING (http://www.squoring.com)
- Mia-Quality by Mia-Software (http://www.mia-software.com)

The results of the tool evaluation in the following section refers to the Sonar implementation of the model. Sonar is an open platform to manage code quality (http://www.sonarsource.org). Using an extensive plug-in mechanism it is fairly easy to extend the basic functionality of the framework (e. g., support analysis for new languages, add new metrics).

The Technical Debt Evaluation (SQALE) Sonar plug-in is a full implementation of the SQALE methodology. This method contains both a Quality Model and an Analysis Model. The Technical Debt Evaluation (SQALE) plug-in comes with a number of features, including custom widgets, visualizations, rules and drill-downs. Figure 3.7 shows the summary page of the tool.

3.4.3 *QUAMOCO Quality Model*

The QUAMOCO framework (see Section 3.2.4.3) is available as an open-source Eclipse extension (https://quamoco.in.tum.de). The Quamoco Consortium provides a toolchain [233] for the creation/editing of quality models and for the automatic analysis of software products:

- Quality Model Editor: This editor enables the comfortable creation of quality models.

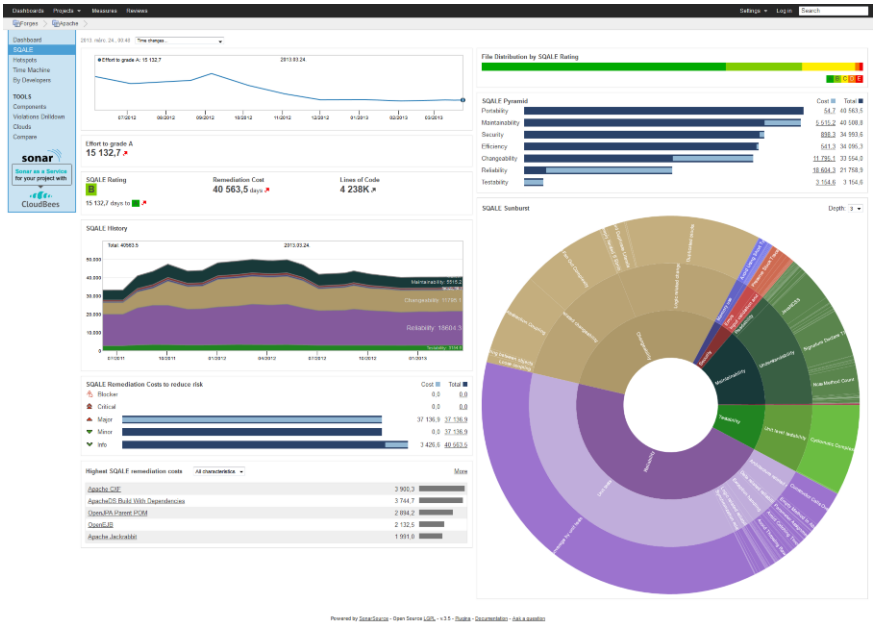

Fig. 3.7: Sonar SQALE Maintainability Model plug-in

- ConQAT-Integration: By integrating the quality model into the analysis framework ConQAT [234], automatic quality assessments for the programming languages Java, C#, and C/C++ can be conducted.

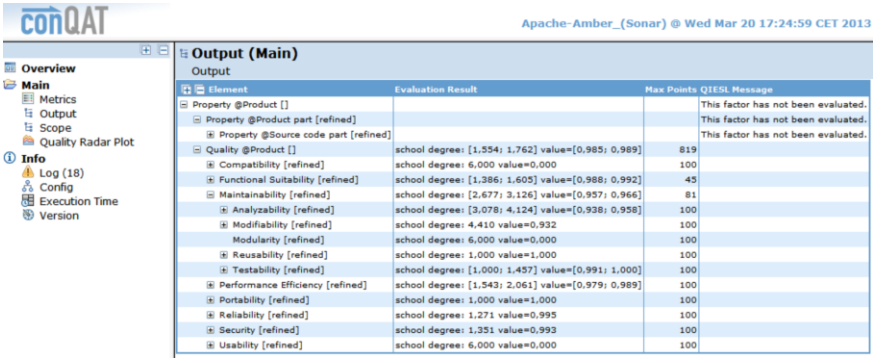

Fig. 3.8: QUAMOCO quality report

The quality analysis with the prepared quality models can be started interactively from Eclipse or run from command line allowing to be integrated into the build

processes. The tool presents its results in Eclipse and also creates a detailed HTML quality report (see Figure 3.8).

3.4.4 SIG Maintainability Model

The Software Improvement Group (http://www.sig.eu) offers software product certification based on the implementation of their maintainability model (see Section 3.2.4.4) as a commercial service. No official downloadable version of the tool exists on their homepage.

However, the SIG Maintainability Model is implemented as a free and downloadable Sonar plug-in. The results of the tool evaluation in the following section refers to this Sonar plug-in implementation of the model. The SIG plug-in provides a high-level overview about the following ISO/IEC 9126 maintainability subcharacteristics: Analysability, Changeability, Stability and Testability. The values range from −− (very bad) to ++ (very good). Figure 3.9 shows a screenshot about the results of the plug-in.

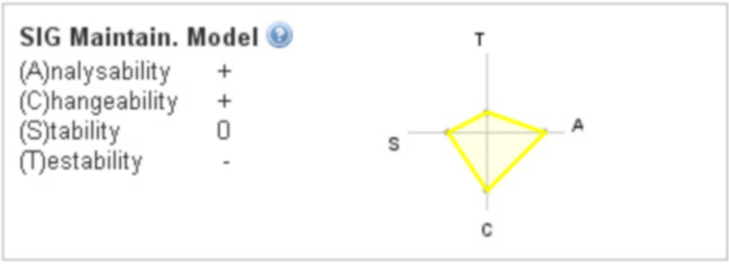

Fig. 3.9: Sonar SIG Maintainability Model Plug-in

3.4.5 Columbus Quality Model

The Columbus Quality Model (see Section 3.2.4.5) is implemented by a proprietary tool called SourceAudit, member of the QualityGate product family. The QualityGate source code quality assurance platform developed by FrontEndART (http://www.frontendart.com) is based on research conducted at the Department of Software Engineering of University of Szeged and on the ISO/IEC 9126 standard.

The tool is able to continuously monitor the maintainability of software products. It can be integrated into the common build processes or manage individual

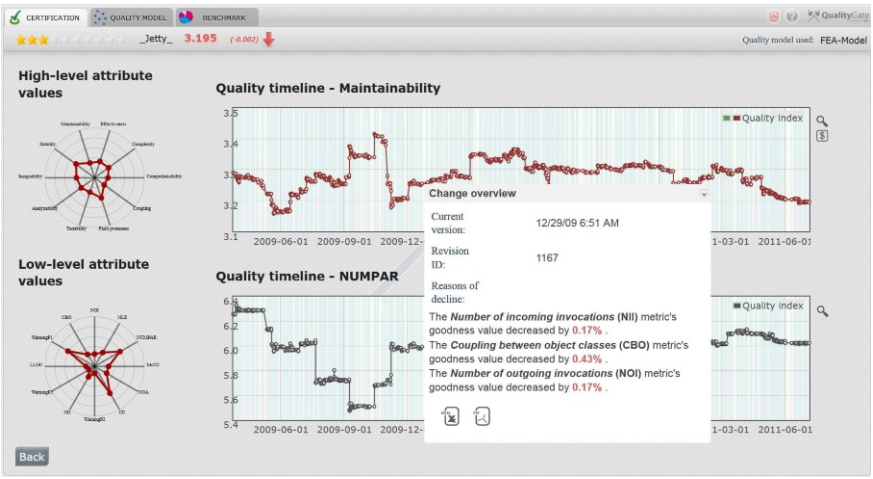

Fig. 3.10: QualityGate implementation of the Columbus Quality Model

qualifications with the help of a Jenkins continuous integration system[1] based administration page. The results of the qualification is presented in a sophisticated web application but many types of different reports can also be generated. Figure 3.10 shows a screenshot of the tool containing a one year long period of quality analysis results.

3.5 Comparing the Features of the Quality Models and Tools

To evaluate and compare the different models and their implementing tools, we installed and ran them on several projects. As two of the tools were available as Sonar plugins, we decided to perform a maintainability assessment on the open-source projects presented in Sonar's Nemo demo application.[2] The benefit of it was twofold: the data in Nemo already contained the quality analysis results of the SQALE model commercial plug-in; and we could easily identify the exact source code locations and versions from Sonar to be able to run the other tools on the same source code.

As the SIG model is not part of Nemo, we also installed and configured our own local version of Sonar. Besides the SQALE and SIG models we decided to include the Sonar Quality Index plug-in[3] in the evaluation as well. It is a Maintainability Index style combination of different metrics and not a hierarchical quality model. Nonetheless, we were interested in the relation of QI to other sophisticated models.

[1] http://jenkins-ci.org/

[2] http://nemo.sonarsource.org/

[3] http://docs.codehaus.org/display/SONAR/Quality+Index+Plugin

Altogether 97 open-source Java projects have been analyzed with six tools. Although Nemo contains almost 200 systems, 50 of them do not have any version control information, therefore we could not locate their source. For around another 50 projects the version control information was changed since the Sonar analysis, so we also left them out from the experiment. Except SQUALE, all the analyses have been run in an automated way with default models and configurations. In the case of SQUALE, we found no way of automating the qualification process, therefore all the projects have been configured and analyzed manually through its web interface. When a qualification analysis failed, we tried to manually fix the causing problem and re-run the analysis. If a more complex error occurred – which we could not fix easily – we marked the analysis as failed. The typical causes of errors are listed in Section 3.5.2.

3.5.1 Comparing the Properties of Different Practical Models

The comparison of the models was done using the following evaluation criteria:

1. *Interpretability* – applying the model should provide information for high-level quality characteristics which is meaningful, i. e., conclusions can be drawn.
2. *Explicability* – there should be a way to efficiently evaluate the root causes, i. e., a simple way to decompose information obtained for high-level characteristics to attributes or even to properties.
3. *Consistency* – the information obtained for higher level characteristics should not contradict lower level information.
4. *Scalability* – the model should provide valuable information even for large systems in reasonable time.
5. *Extendibility* – there should be an easy way to extend the model with new characteristics and its attributes.
6. *Reproducibility* – applying the model on the same system twice should result in the same information.
7. *Comparability* – information obtained for quality characteristics of two different systems should be comparable and should correlate with the intuitive meaning of the characteristics.
8. *Aggregation type* – the way of acquiring quality values for high-level characteristics based on low-level values. The possible values are:

    - Linear combination (LC) – a simple linear combination of the values.
    - General function (GF) – combination of the values with an arbitrary (not necessarily linear) function.
    - Fixed threshold (FT) – the values are categorized based on fixed thresholds.
    - Benchmark-based threshold (BT) – the values are categorized based on thresholds derived from a benchmark.
    - Benchmark based (B) – the aggregation is done in some sophisticated way based on a repository of other systems (benchmark).

9. *Input measures* – what type of source code measures are considered in the model. The possible values are:
   - Metrics (M)
   - Rule violations (R)
   - Code clones (C)
   - Unit tests (T)
10. *Base model* – which theoretical model serves as the base concept of the practical model.
11. *Rating* – what kind of qualification or rating does the model provide for expressing the level of maintainability. The possible values are:
   - Ordinal – discrete quality categories (like 1 to 5 stars, etc.)
   - Scale – a continuous value from an interval (e. g., a real number between 0 and 10)

Table 3.6 presents the summary of the model evaluations against the criteria above. We can note that the most popular base model is the one defined in the ISO/IEC 9126 standard. Despite the fact that it already has a successor – ISO/IEC 25000 – only one model supports it in some extent. Probably most of the models will adapt to the new standard in the future.

Table 3.6: The properties of the different practical quality models

	SQALE	**ColumbusQM**	**SIG**	**QI**	**SQUALE**	**QUAMOCO**
Interpretable	✓	✓	✓	✓	✓	✓
Explicable	✓	✓	–[4]	–	✓	✓
Consistent	✓	✓	✓	✓	✓	✓
Scalable	N/A[5]	✓	✓	✓	✓[6]	✓
Extendible	–	✓	–	–	–	✓
Reproducible	✓	✓	✓	✓	✓	✓
Comparable	✓	✓	✓	✓	✓	✓
Aggregation type	FT	B	BT	LC	FT+GF	FT
Input measures	M, R	M, R, C	M, C, T	M, R, C, T	M, R	M, R
Base model	ISO 9126	ISO 9126	ISO 9126		McCall, ISO 9126	partly ISO 9126, ISO 25000
Rating	Ordinal A, B, C, D, E	Scale [0..10]	Ordinal [-2..2]	Scale [0..10]	Scale [0..3]	Scale [1..6]

Regarding the rating of the models, the scale type appears to be in majority which is able to express the maintainability in a more precise, continuous way. Another

[4] Refers to the Sonar plug-in which does not allow to drill-down the qualifications

[5] SQALE qualifications were already available in Sonar Nemo

[6] We found performance issues with the default embedded database, but we have not tried with other suggested database servers

advantage of the scale type ratings is that it is easy to convert the rating of one model to the rating of the other. On the contrary, ordinal ratings are harder to convert due to the different number of rates.

One would expect that a model uses all the possible static source code information: metrics, rule violations, code clones and unit tests as its input measures. However, only the Quality Index seams to use all this information. Metrics are considered by all the examined models and rule violations are taken into account by all models except SIG.

The models vary in the way they aggregate the source code measures. The most common approach is to use a fixed threshold to categorize metric values. However, a constant improvement is shown in this area by introducing complex aggregation formulas and deriving dynamic thresholds based on a benchmark. The ColumbusQM uses the benchmark in even a more sophisticated way to aggregate quality properties.

Almost all of our requirements are met by the examined models. Most of the models failed to fulfill the Extendibility requirements as they provide no easy way to extend the base model. Another requirement that two models could not met is Explicability. The results of the models that do not fulfil this requirement is hard to be traced back to the root causes in the source code.

3.5.2 Evaluating the Properties of the Different Tools

The evaluation of the tools was made according to the following aspects:

1. *Supported languages* – the languages supported by the tool (or it is language independent).
2. *Stability* – the number of projects successfully analyzed from all projects (in total 97 projects have been analyzed).
3. *Input type* – the input of the tool i. e., requires only sources or binaries too.
4. *Type* – the type of the application (e. g., a plug-in to an existing framework, a web application).
5. *Supported build processes* – the type of common build frameworks into which the qualification can be integrated.
6. *OS platform* – the supported OS platforms.
7. *Proprietary* – is the evaluated tool free or proprietary?
8. *Presentation of the results* – the way of the presentation of qualification results (e. g., in a web application, HTML)

Table 3.7 presents the summary of the evaluation of the tools against the aspects above. The stability line needs some further explanation. In case of SQALE all the projects were successfully analyzed because it was already in the Sonar Nemo system. The other tool that was able to parse all the systems is QualityGate SourceAudit, because it is able to analyze projects without having to compile the code. In the case of the two Sonar plugins, the SIG model and Quality Index, the cause of

Table 3.7: The properties of the different evaluated tools

	SQALE	**QualityGate SourceAudit**	**SIG**
Supported languages	Lang. independent	Lang. independent	Lang. independent
Stability	100% (97/97)	100% (97/97)	77% (75/97)
Input type	Sources, binaries are optional	Sources only	Sources, binaries are optional
Type	Sonar plug-in	Web application and web service	Sonar plug-in
Supported build processes	ant, maven, batch	ant, maven, batch	ant, maven, batch
OS platform	Windows & Linux	Windows & Linux	Windows & Linux
Proprietary	Yes	Yes	Yes (free Sonar plugin)
Presentation of the results	Web application	Web application, Excel, PDF reports	Web application

	QI	**SQUALE**	**QUAMOCO**
Supported languages	Java	Java	Java, C#, and C/C++
Stability	77% (75/97)	31% (30/97)	63% (61/97)
Input type	Sources, binaries are optional	Sources and binaries	Sources and binaries
Type	Sonar plug-in	Web application	Eclipse plug-in
Supported build processes	ant, maven, batch	ant	batch
OS platform	Windows & Linux	Windows & Linux	Windows & Linux
Proprietary	No	No	No
Presentation of the results	Web application	Web application, PDF reports	Eclipse GUI, HTML report

unsuccessful qualification was that some of the projects could not be compiled and not the failure of the models. As we used the maven wrapper to upload the results into Sonar, it caused the failure of the qualification also. The other two tools, QUAMOCO and SQUALE are also affected by the compilation errors as they require the binaries for the qualification. Additionally to the build errors, QUAMOCO failed with a non-trivial parser error for about 10 projects. The most unstable tool was SQUALE according to our experiences; however, it must be noted that we used the program with default settings only.

To summarize, most tools were able to analyze the majority of the projects with minimal invested effort. Therefore, they can be a great help both for managers and developers in software evolution activities.

3.6 Conclusions

For software developers and managers alike it is crucial to be able to measure different aspects of the quality of their systems. The information can mainly be used for making decisions, backing up intuition, estimating future costs and assessing risks during software evolution.

There are three main approaches for measuring software quality: process-based, product-based and hybrid. This chapter focused on the history, evolution, state-of-the-art and supporting tools of the product based software quality assessment. The introduction of the ISO/IEC 9126 standard as the joint model of the early theoretical software product quality models caused an explosion in the number of new practical quality models. All these models adapt the standard and use a hierarchical model for estimating quality with some kind of metrics at the lowest level. Section 3.2.4 gives an overview about the evolution of software quality measurements and approaches starting from the first software metrics through simple metrics-based prediction models and early theoretical quality models to focus on the currently available state-of-the-art approaches for software product qualification.

Each of the tools implementing these models have been evaluated on almost 100 open-source Java systems. The tools and underlying quality models were compared according to a set of predefined criteria. Most tools were able to analyze the majority of the projects with minimal invested effort. Therefore, we conclude that they can be a great help both for managers and developers in software evolution activities. However, we note that the correctness of the models has not been evaluated. In the end irrespective of how easy it is to use them or what features they have, we expect that models that are more accurate will be more frequently used. However, comparing the correctness of the existing models requires a huge effort that should be addressed by the joint work of the community.

It is also a very interesting open question if the state-of-the-art practical models can be unified and merged into a common standard, like it was done with the early theoretical models. To be able to assess this possibility, a very deep analysis of model results would be needed. It should be examined how well the results of the current practical models correlate with each other. Our vision is that these practical models can be merged into a common standard in the future which will lead to a more exact and objective product quality assessment.